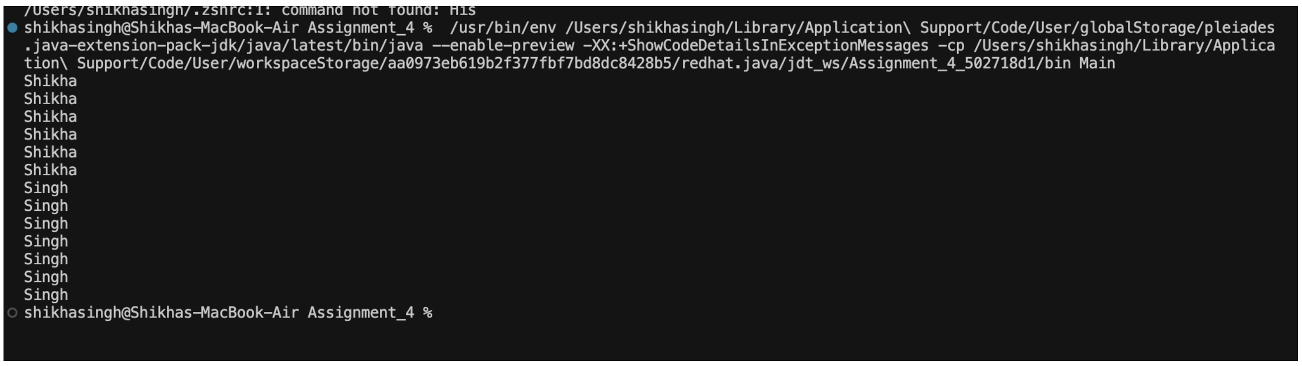
Java Assignment 4

Name: Shikha Singh

Roll no: 25

Q1. WAP to create two threads by extending Thread class. Print your "first name" in thread-1 six times and Print your "last name" in thread-2 seven times.

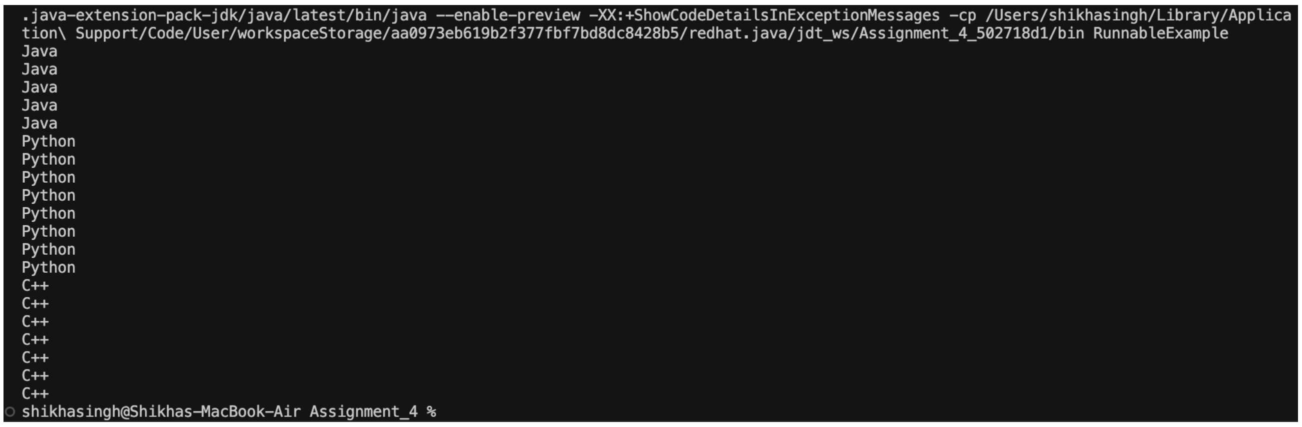
```
class FirstNameThread extends Thread {
   public void run() {
      for (int i = 0; i < 6; i++) {
         System.out.println("Shikha");
      }
   }
}
class LastNameThread extends Thread {
   public void run() {
      for (int i = 0; i < 7; i++) {
         System.out.println("Singh");
      }
   }
}
public class Main {
   public static void main(String[] args) {
      FirstNameThread thread1 = new FirstNameThread();
      LastNameThread thread2 = new LastNameThread();

      thread1.start();
      thread2.start();
   }
}
class JavaThread implements Runnable {
   public void run() {
      for (int i = 0; i < 5; i++) {
         System.out.println("Java");
      }
   }
}
class PythonThread implements Runnable {
   public void run() {
      for (int i = 0; i < 8; i++) {
         System.out.println("Python");
      }
   }
}
class CppThread implements Runnable {
    public void run() {
        for (int i = 0; i < 7; i++) {
            System.out.println("C++");
        }
    }
}
class RunnableMain {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new JavaThread());
        Thread thread2 = new Thread(new PythonThread());
        Thread thread3 = new Thread(new CppThread());

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```



```
/Users/shikhasingh/.zshrc:1: command not found: His
● shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globalStorage/pleiades
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Applica
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin Main
Shikha
Shikha
Shikha
Shikha
Shikha
Shikha
Singh
Singh
Singh
Singh
Singh
Singh
Singh
○ shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

Q2. WAP to create three threads by implementing Runnable interface. Print “Java” in thread-1 five times and Print your “Python” in thread-2 eight times. Print your “C++” in thread-3 seven times.

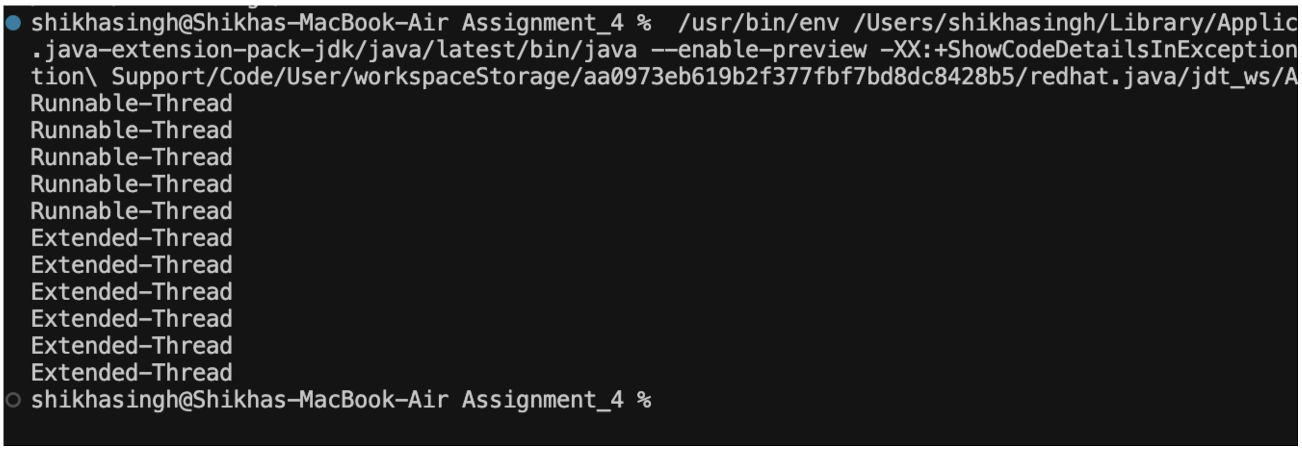
```
class JavaThread implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Java");
        }
    }
}
class PythonThread implements Runnable {
    public void run() {
        for (int i = 0; i < 8; i++) {
            System.out.println("Python");
        }
    }
}
class CppThread implements Runnable {
    public void run() {
        for (int i = 0; i < 7; i++) {
            System.out.println("C++");
        }
    }
}
public class RunnableExample {
    public static void main(String[] args) {

        Thread thread1 = new Thread(new JavaThread());
        Thread thread2 = new Thread(new PythonThread());
        Thread thread3 = new Thread(new CppThread());

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```



```
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Applica
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin RunnableExample
Java
Java
Java
Java
Java
Python
Python
Python
Python
Python
Python
Python
Python
C++
C++
C++
C++
C++
C++
C++
○ shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

Q3. WAP to create two threads. For creating thread-1, implement Runnable interface. For creating thread-2, extend thread class. Print name of thread in thread-1 five times and Print name of thread in thread-2 six times. Use the concept of anonymous class to implement Runnable interface.

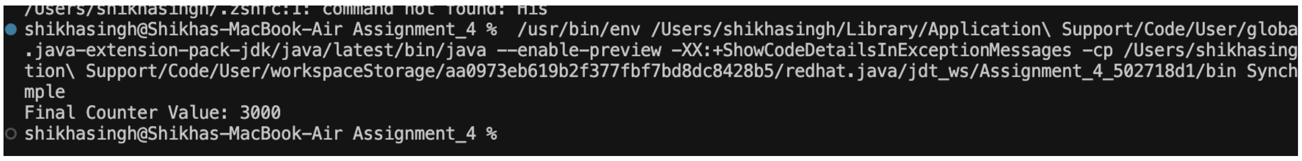
```
class Thread2 extends Thread {
    public void run() {
        for (int i = 0; i < 6; i++) {
            System.out.println(Thread.currentThread().getName());
        }
    }
}
public class ThreadExample {
    public static void main(String[] args) {
        Runnable thread1 = new Runnable() {
            public void run() {
                for (int i = 0; i < 5; i++) {
                    System.out.println(Thread.currentThread().getName());
                }
            }
        };

        Thread t1 = new Thread(thread1);
        Thread2 t2 = new Thread2();

        t1.setName("Runnable-Thread");
        t2.setName("Extended-Thread");

        t1.start();
        t2.start();
    }
}
```



```
● shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Applic
  .java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInException
  tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/A
  Runnable-Thread
  Runnable-Thread
  Runnable-Thread
  Runnable-Thread
  Runnable-Thread
  Extended-Thread
  Extended-Thread
  Extended-Thread
  Extended-Thread
  Extended-Thread
  Extended-Thread
○ shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

Q4. Write a Java program to create and start multiple threads that increment a shared counter variable concurrently. (You should get correct result.) (Hint: use the concept of synchronized method).

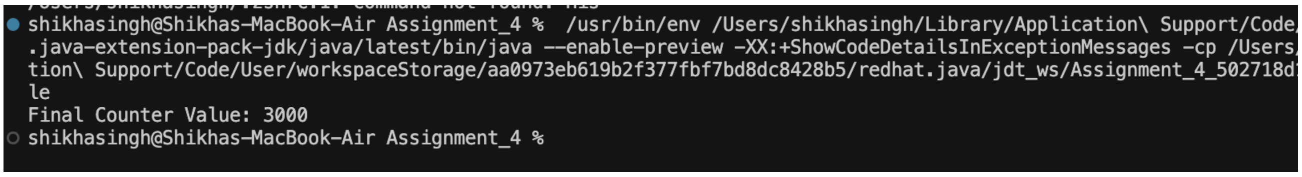
```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
class CounterThread extends Thread {
    private Counter counter;

    public CounterThread(Counter counter) {
        this.counter = counter;
    }
    public void run() {

        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
public class SynchronizedCounterExample {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread thread1 = new CounterThread(counter);
        Thread thread2 = new CounterThread(counter);
        Thread thread3 = new CounterThread(counter);

        thread1.start();
        thread2.start();
        thread3.start();
        try {

            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final Counter Value: " + counter.getCount());
    }
}
```



```
/Users/shikhasingh/.zshrc:1: command not found: His
shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globa
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasing
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin Synch
mple
Final Counter Value: 3000
shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

Q5. Write a Java program to demonstrate the concept of synchronized blocks.

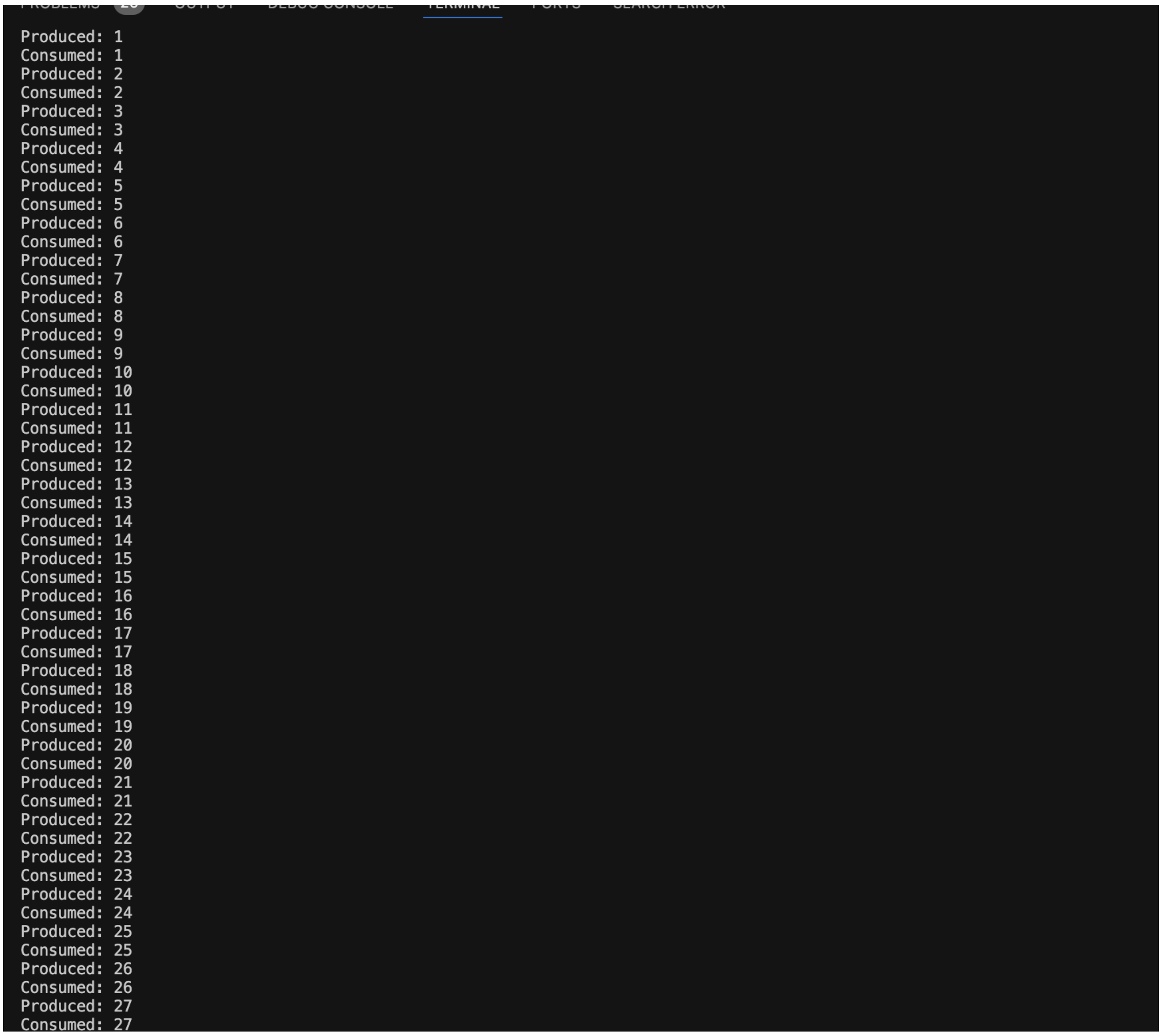
```
class Counter {
    private int count = 0;
    public void increment() {
        synchronized (this) {
            count++;
        }
    }
    public int getCount() {
        return count;
    }
}
class CounterThread extends Thread {
    private Counter counter;
    public CounterThread(Counter counter) {
        this.counter = counter;
    }
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
public class SynchronizedBlockExample {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread thread1 = new CounterThread(counter);
        Thread thread2 = new CounterThread(counter);
        Thread thread3 = new CounterThread(counter);
        thread1.start();
        thread2.start();
        thread3.start();
        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final Counter Value: " + counter.getCount());
    }
}
```



```
shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d
le
Final Counter Value: 3000
shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

Q6. Write a Java program to create a producer-consumer scenario using the wait() and notify() methods for thread synchronization.

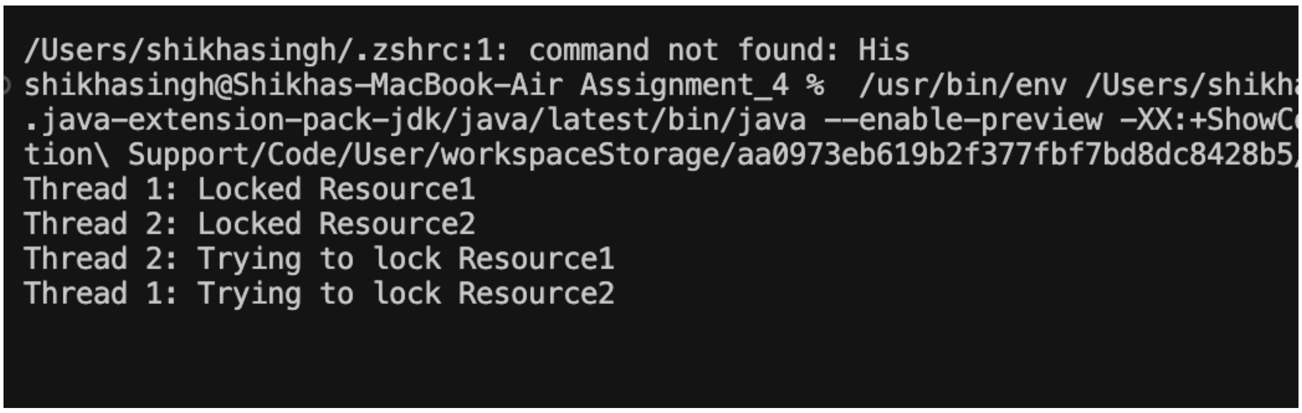
```
class SharedResource {
    private int item;
    private boolean available = false;
    public synchronized void produce(int value) throws InterruptedException {
        while (available) {
            wait();
        }
        item = value;
        available = true;
        System.out.println("Produced: " + item);
        notify();
    }
    public synchronized void consume() throws InterruptedException {
        while (!available) {
            wait();
        }
        System.out.println("Consumed: " + item);
        available = false;
        notify();
    }
}
class Producer extends Thread {
    private SharedResource resource;
    public Producer(SharedResource resource) {
        this.resource = resource;
    }
    public void run() {
        int value = 0;
        while (true) {
            try {
                resource.produce(++value);
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class Consumer extends Thread {
    private SharedResource resource;
    public Consumer(SharedResource resource) {
        this.resource = resource;
    }
    public void run() {
        while (true) {
            try {
                resource.consume();
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class ProducerConsumerExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        Producer producer = new Producer(resource);
        Consumer consumer = new Consumer(resource);
        producer.start();
        consumer.start();
    }
}
```



```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
Produced: 10
Consumed: 10
Produced: 11
Consumed: 11
Produced: 12
Consumed: 12
Produced: 13
Consumed: 13
Produced: 14
Consumed: 14
Produced: 15
Consumed: 15
Produced: 16
Consumed: 16
Produced: 17
Consumed: 17
Produced: 18
Consumed: 18
Produced: 19
Consumed: 19
Produced: 20
Consumed: 20
Produced: 21
Consumed: 21
Produced: 22
Consumed: 22
Produced: 23
Consumed: 23
Produced: 24
Consumed: 24
Produced: 25
Consumed: 25
Produced: 26
Consumed: 26
Produced: 27
Consumed: 27
```

Q7. WAP to demonstrate deadlock among multiple threads.

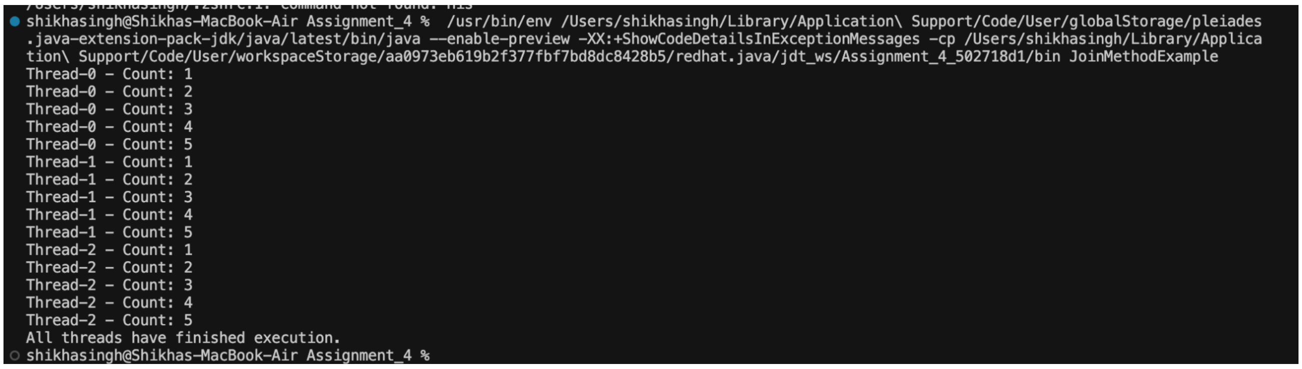
```
class Resource1 {
   public synchronized void method1(Resource2 resource2) {
      System.out.println("Thread 1: Locked Resource1");

      try {
         Thread.sleep(100);
      } catch (InterruptedException e) {}

      System.out.println("Thread 1: Trying to lock Resource2");
      resource2.method2();
   }

   public synchronized void method2() {
        System.out.println("Thread 1: Locked Resource1 again");
    }
}

class Resource2 {
    public synchronized void method1(Resource1 resource1) {
        System.out.println("Thread 2: Locked Resource2");

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}

        System.out.println("Thread 2: Trying to lock Resource1");
        resource1.method2();
    }

    public synchronized void method2() {
        System.out.println("Thread 2: Locked Resource2 again");
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        final Resource1 resource1 = new Resource1();
        final Resource2 resource2 = new Resource2();

        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                resource1.method1(resource2);
            }
        });

        Thread thread2 = new Thread(new Runnable() {
            public void run() {
                resource2.method1(resource1);
            }
        });

        thread1.start();
        thread2.start();
    }
}
```



```
/Users/shikhasingh/.zshrc:1: command not found: His
shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikh
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowC
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5
Thread 1: Locked Resource1
Thread 2: Locked Resource2
Thread 2: Trying to lock Resource1
Thread 1: Trying to lock Resource2
```

Q8. WAP to demonstrate the working of join() method.

```
class JoinThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - Count: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class JoinMethodExample {
    public static void main(String[] args) {
        JoinThread thread1 = new JoinThread();
        JoinThread thread2 = new JoinThread();
        JoinThread thread3 = new JoinThread();
        thread1.start();
        try {
            thread1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread2.start();
        try {
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread3.start();
        try {
            thread3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("All threads have finished execution.");
    }
}
```

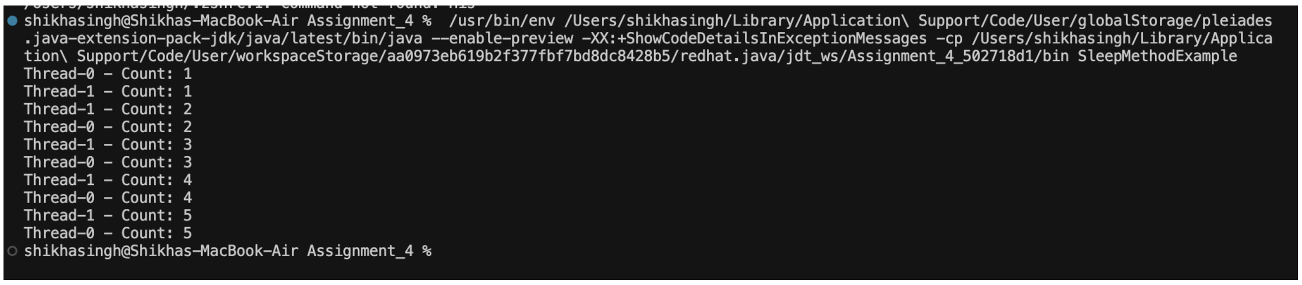


```
shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globalStorage/pleiades
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Applica
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin JoinMethodExample
Thread-0 - Count: 1
Thread-0 - Count: 2
Thread-0 - Count: 3
Thread-0 - Count: 4
Thread-0 - Count: 5
Thread-1 - Count: 1
Thread-1 - Count: 2
Thread-1 - Count: 3
Thread-1 - Count: 4
Thread-1 - Count: 5
Thread-2 - Count: 1
Thread-2 - Count: 2
Thread-2 - Count: 3
Thread-2 - Count: 4
Thread-2 - Count: 5
All threads have finished execution.
shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

Q9. WAP to demonstrate the working of Thread.sleep() method.

```
class SleepThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - Count: " + i);
            try {
                Thread.sleep(1000); // Sleep for 1 second (1000 milliseconds)
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}


public class SleepMethodExample {
    public static void main(String[] args) {
        SleepThread thread1 = new SleepThread();
        SleepThread thread2 = new SleepThread();


        thread1.start();
```

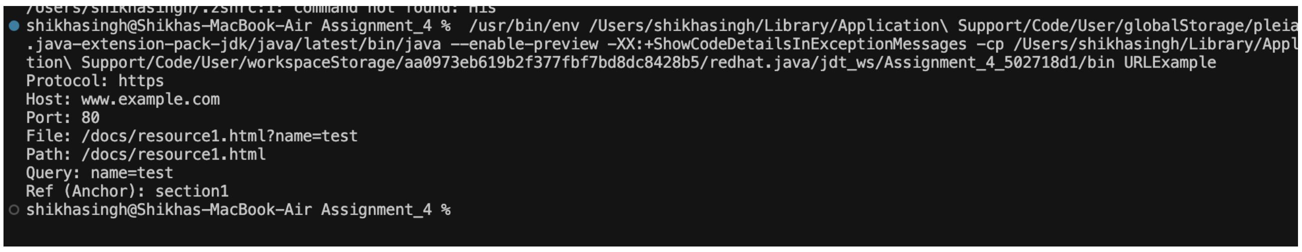
```java
        thread2.start();
    }
}
```



```
shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globalStorage/pleiades
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Applica
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin SleepMethodExample
Thread-0 - Count: 1
Thread-1 - Count: 1
Thread-1 - Count: 2
Thread-0 - Count: 2
Thread-1 - Count: 3
Thread-0 - Count: 3
Thread-1 - Count: 4
Thread-0 - Count: 4
Thread-1 - Count: 5
Thread-0 - Count: 5
shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

Q10. WAP to demonstrate URL class.

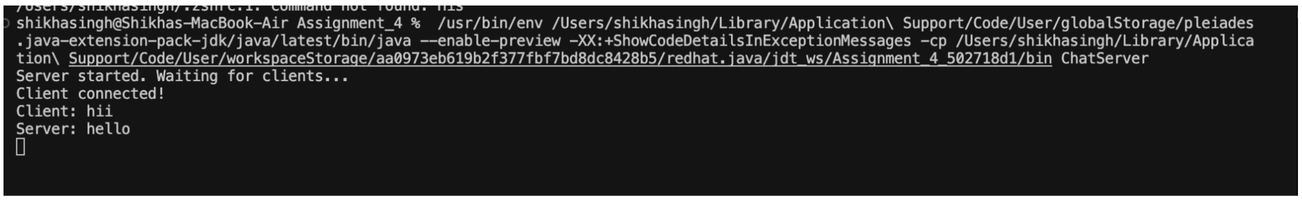
```java
import java.net.URL;
import java.net.MalformedURLException;


public class URLExample {
    public static void main(String[] args) {
        try {
            URL url = new
URL("https://www.example.com:80/docs/resource1.html?name=test#section1");


            System.out.println("Protocol: " + url.getProtocol());
            System.out.println("Host: " + url.getHost());
            System.out.println("Port: " + url.getPort());
            System.out.println("File: " + url.getFile());
            System.out.println("Path: " + url.getPath());
            System.out.println("Query: " + url.getQuery());
            System.out.println("Ref (Anchor): " + url.getRef());
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```



```
/Users/shikhasingh/.zshrc:1: command not found: his
● shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globalStorage/pleia
  .java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Appl
  tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin URLExample
  Protocol: https
  Host: www.example.com
  Port: 80
  File: /docs/resource1.html?name=test
  Path: /docs/resource1.html
  Query: name=test
  Ref (Anchor): section1
○ shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

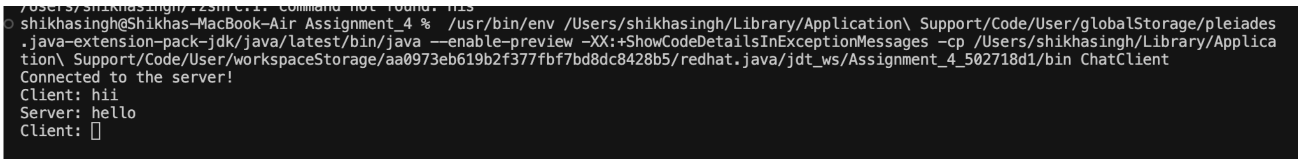
Q11. WAP to create chat application using the concept of connection oriented approach using TCP protocol.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
public class ChatServer {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8080);
            System.out.println("Server started. Waiting for clients...");
            Socket socket = serverSocket.accept();
            System.out.println("Client connected!");

            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

            BufferedReader serverInput = new BufferedReader(new InputStreamReader(System.in));
            String clientMessage, serverMessage;

            while (true) {
                clientMessage = in.readLine();
                if (clientMessage.equalsIgnoreCase("bye")) {
                    System.out.println("Client disconnected.");
                    break;
                }
                System.out.println("Client: " + clientMessage);
                System.out.print("Server: ");
                serverMessage = serverInput.readLine();
                out.println(serverMessage);
                if (serverMessage.equalsIgnoreCase("bye")) {
                    System.out.println("Server shutting down.");
                    break;
                }
            }
            socket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```
shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globalStorage/pleiades
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Applica
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin ChatServer
Server started. Waiting for clients...
Client connected!
Client: hii
Server: hello
```

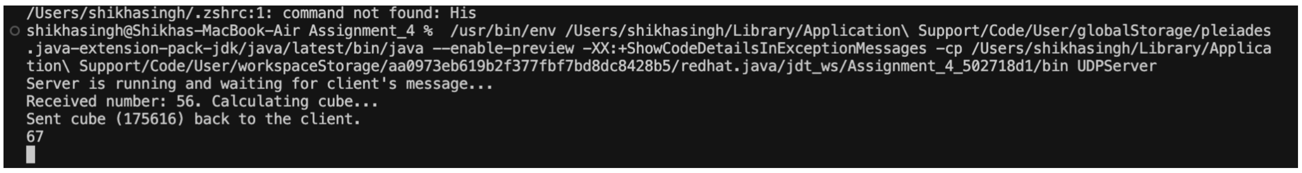
```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
public class ChatClient {
  public static void main(String[] args) {
    try {
        Socket socket = new Socket("localhost", 8080);
        System.out.println("Connected to the server!");

        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

        BufferedReader clientInput = new BufferedReader(new InputStreamReader(System.in));
        String clientMessage, serverMessage;

        while (true) {
            System.out.print("Client: ");
            clientMessage = clientInput.readLine();
            out.println(clientMessage);
            if (clientMessage.equalsIgnoreCase("bye")) {
                System.out.println("Disconnected from the server.");
                break;
            }
            serverMessage = in.readLine();
            System.out.println("Server: " + serverMessage);
            if (serverMessage.equalsIgnoreCase("bye")) {
                System.out.println("Server disconnected.");
                break;
            }
        }
        socket.close();
    } catch (IOException e) {
        e.printStackTrace();
        }
    }
}
```



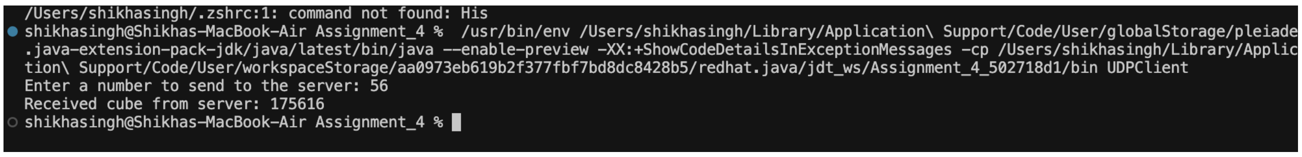
```
/Users/shikhasingh/.zshrc:1: command not found: his
shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globalStorage/pleiades
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Applica
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin ChatClient
Connected to the server!
Client: hii
Server: hello
Client: 
```

Q12. WAP to create a UDP client application and UDP server application. Client will send a number to server. Server has to calculate the cube of a number sent back to the client.

```
import java.net.*;
public class UDPServer {
    public static void main(String[] args) {
        try {
            DatagramSocket serverSocket = new DatagramSocket(9876);
            byte[] receiveData = new byte[1024];
            byte[] sendData;
            System.out.println("Server is running and waiting for client's message...");
            while (true) {

                DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
                serverSocket.receive(receivePacket);
                String numberString = new String(receivePacket.getData(), 0, receivePacket.getLength());

                int number = Integer.parseInt(numberString);
                int cube = number * number * number;

                System.out.println("Received number: " + number + ". Calculating cube...");

                String resultString = Integer.toString(cube);
                sendData = resultString.getBytes();

                InetAddress clientAddress = receivePacket.getAddress();
                int clientPort = receivePacket.getPort();

                DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, clientAddress, clientPort);
                serverSocket.send(sendPacket);
                System.out.println("Sent cube (" + cube + ") back to the client.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


```
/Users/shikhasingh/.zshrc:1: command not found: His
○ shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globalStorage/pleiades
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Applica
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin UDPServer
Server is running and waiting for client's message...
Received number: 56. Calculating cube...
Sent cube (175616) back to the client.
67
```

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
public class UDPClient {
  public static void main(String[] args) {
    try {
      BufferedReader userInput = new BufferedReader(new InputStreamReader(System.in));
      DatagramSocket clientSocket = new DatagramSocket();
      InetAddress serverAddress = InetAddress.getByName("localhost");
      byte[] sendData;
      byte[] receiveData = new byte[1024];
      System.out.print("Enter a number to send to the server: ");
      String numberString = userInput.readLine();
      sendData = numberString.getBytes();
      DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, serverAddress, 9876);
      clientSocket.send(sendPacket);
      DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
      clientSocket.receive(receivePacket);
      String cubeResult = new String(receivePacket.getData(), 0, receivePacket.getLength());
      System.out.println("Received cube from server: " + cubeResult);
      clientSocket.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```


```
/Users/shikhasingh/.zshrc:1: command not found: His
● shikhasingh@Shikhas-MacBook-Air Assignment_4 %  /usr/bin/env /Users/shikhasingh/Library/Application\ Support/Code/User/globalStorage/pleiade
.java-extension-pack-jdk/java/latest/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMessages -cp /Users/shikhasingh/Library/Applic
tion\ Support/Code/User/workspaceStorage/aa0973eb619b2f377fbf7bd8dc8428b5/redhat.java/jdt_ws/Assignment_4_502718d1/bin UDPClient
Enter a number to send to the server: 56
Received cube from server: 175616
○ shikhasingh@Shikhas-MacBook-Air Assignment_4 %
```

Q13. Differentiate between process and thread?

Aspect	Process	Thread
Definition	A process is an independent program in execution, with its own memory and resources.	A thread is a smaller unit of a process that runs within the process and shares its resources.
Memory	Each process has its own separate memory space (heap, stack, data segments).	Threads share the same memory space of the process they belong to.

Communication	Inter-process communication (IPC) is required for processes to communicate (e.g., pipes, sockets).	Threads can directly communicate with each other by accessing shared variables and resources.
Overhead	Creating a process is heavier and more resource-intensive.	Creating threads is lighter and requires fewer resources.
Context Switching	Process context switching is more expensive due to separate memory and resources.	Thread context switching is faster as threads share memory within the same process.
Independence	Processes are independent of each other, and failure in one process generally doesn't affect others.	Threads within the same process are interdependent, and a failure in one thread can affect others within the same process.
Concurrency	Processes can execute concurrently but typically require more OS overhead for multitasking.	Threads allow for more fine-grained parallelism and concurrency within the same process.
Isolation	Each process is isolated from others, offering better security and protection.	Threads share the same address space, so they are less isolated and more prone to issues like race conditions.
Creation	Processes are created using system calls like fork() in Unix or CreateProcess() in Windows.	Threads are created using libraries or language features such as Thread in Java or pthread in C/C++.
Termination	When a process terminates, all associated threads also terminate.	When a thread terminates, the process can continue to run if other threads are still active.
Example	Running two separate programs like a web browser and a text editor.	Running multiple tasks within a program, like downloading a file and rendering a webpage in a web browser.

Key Points:

- **Processes** are heavier, independent units of execution, often used when isolation between tasks is necessary.
- **Threads** are lighter, more efficient, and share resources, making them suitable for tasks that need to run concurrently within the same application.

Q14. Differentiate between Process based multitasking and thread based multitasking

Aspect	Process-Based Multitasking	Thread-Based Multitasking
Definition	Involves running multiple independent processes concurrently.	Involves running multiple threads within the same process concurrently.
Memory	Each process has its own separate memory space and resources.	Threads within the same process share the same memory and resources.
Communication	Processes require Inter-Process Communication (IPC) mechanisms like pipes, sockets, or shared memory for	Threads can communicate easily by accessing shared memory within the same process.

	communication.	
Resource Usage	Higher resource usage since each process requires its own memory and system resources.	Lower resource usage as threads share the same memory and resources of the parent process.
Context Switching	Context switching between processes is slower due to the need to save and load process-specific memory, registers, and resources.	Context switching between threads is faster as threads share the same memory and resources.
Overhead	Creating and managing processes is more resource-intensive and involves higher overhead.	Threads are lightweight, and creating/managing threads involves much lower overhead.
Concurrency	Processes run independently, offering more secure and isolated multitasking, but with less concurrency due to high overhead.	Threads allow for higher concurrency and better responsiveness due to their lightweight nature and shared memory.
Fault Isolation	If one process crashes, it generally does not affect other processes.	If a thread crashes, it can potentially impact the entire process and other threads within it.
Security	Processes are more secure and isolated, as they don't share memory.	Threads have less security due to shared memory, making them more susceptible to issues like race conditions and deadlocks.
Example	Running multiple programs like a web browser, a video editor, and a music player simultaneously.	Running multiple tasks in a web browser, like loading different tabs or rendering a page while downloading files.
Use Case	Best suited for situations where tasks need to be isolated from each other, or when different programs are running independently.	Best suited for tasks within the same application that need to perform different operations simultaneously or asynchronously.

Key Points:

- **Process-based multitasking**:
    - Suitable for **isolated tasks**.
    - Has higher **resource overhead** due to independent memory allocation.
    - Offers **strong isolation**, making it more secure but slower.
- **Thread-based multitasking**:
    - Suitable for **concurrent tasks** within the same application.
    - Has **low overhead** and **faster context switching**.
    - Threads share memory, which improves performance but can lead to potential synchronization issues (e.g., race conditions).

Q15. What do you understand by inter-thread communication?

Inter-thread communication refers to the mechanisms that allow threads to communicate and coordinate their actions within a multi-threaded application. Since threads share the same memory space in a process, they can share data and send signals to one another, facilitating efficient collaboration. Here are key aspects of inter-thread communication:

Key Concepts:

1. **Shared Resources**:
    - Threads can access shared variables and data structures. Proper synchronization is essential to avoid data inconsistencies or race conditions.

2. **Synchronization**:
    - Synchronization mechanisms (like synchronized methods or blocks in Java, or mutexes in C/C++) ensure that only one thread can access a resource at a time, preventing conflicts and ensuring data integrity.
3. **Wait/Notify Mechanism**:
    - Threads can use methods like wait(), notify(), and notifyAll() (in Java) to signal between each other.
        - **wait()**: A thread can wait until another thread invokes notify() or notifyAll().
        - **notify()**: Wakes up a single thread that is waiting on the object's monitor.
        - **notifyAll()**: Wakes up all threads waiting on the object's monitor.
4. **Producer-Consumer Problem**:
    - A classic example of inter-thread communication where one thread (the producer) generates data and another thread (the consumer) processes it. Proper synchronization ensures that the producer doesn't add data to a full buffer and the consumer doesn't attempt to consume from an empty buffer.
5. **Thread Coordination**:
    - Threads can communicate about their states or results, allowing one thread to wait for another to complete a task before proceeding.

Benefits:

- **Efficiency**: Inter-thread communication enables efficient data sharing and synchronization, allowing threads to collaborate effectively.
- **Responsiveness**: Threads can signal each other to wake up or perform actions based on shared conditions, enhancing the responsiveness of applications.

Challenges:

- **Deadlocks**: Poorly managed inter-thread communication can lead to deadlocks, where threads wait indefinitely for each other to release resources.
- **Race Conditions**: Without proper synchronization, multiple threads might modify shared data concurrently, leading to inconsistent or erroneous results.

Example:

In a multi-threaded application, a worker thread may process tasks added to a shared queue by a producer thread. The worker thread can wait if the queue is empty (using wait()) and can be notified to resume processing when a new task is added (using notify()).
Overall, inter-thread communication is crucial for ensuring that threads work together seamlessly and efficiently in a concurrent environment.

Q16. What is multithreading. What are the advantages of multithreading?

Multithreading is a programming and execution model that allows multiple threads to run concurrently within a single process. Each thread represents a separate path of execution, enabling the application to perform multiple operations simultaneously. This is particularly useful for improving application performance and responsiveness.

Advantages of Multithreading:

1. **Improved Performance**:
    - Multithreading allows multiple threads to execute simultaneously, particularly on multi-core processors, which can significantly enhance performance for CPU-bound tasks.
2. **Better Resource Utilization**:

- Threads share the same memory space and resources of a process, which reduces the overhead of creating new processes and leads to efficient resource management.

3. **Increased Responsiveness**:
   - Applications can remain responsive to user input while performing background tasks (e.g., downloading files, processing data), as these tasks can run on separate threads.
4. **Simplified Program Structure**:
   - Some problems can be more naturally expressed as concurrent tasks, making the program structure simpler and easier to understand.
5. **Concurrency**:
   - Multithreading enables concurrent execution of tasks, which is beneficial for applications that require simultaneous operations, such as web servers handling multiple client requests.
6. **Scalability**:
   - Multithreaded applications can easily scale by distributing tasks across multiple threads, making them well-suited for modern multi-core and multi-processor systems.
7. **Enhanced Throughput**:
   - By performing multiple operations in parallel, multithreading can increase the overall throughput of an application, leading to faster completion of tasks.
8. **Efficient Communication**:
   - Since threads share the same memory space, communication between them is typically faster and simpler than inter-process communication.

Overall, multithreading is a powerful tool that enhances application efficiency, performance, and user experience by allowing concurrent execution of tasks.

Q17. What is the purpose of join() method?

The join() method in Java is used to ensure that one thread waits for the completion of another thread before it continues its execution. Here's a detailed explanation of its purpose and usage:

Purpose of the join() Method:

1. **Thread Synchronization**:
   - The join() method allows one thread to pause its execution until another thread has finished executing. This is useful for synchronizing the flow of execution between multiple threads.
2. **Ensuring Order of Execution**:
   - In situations where the order of execution matters, you can use join() to make sure that a particular thread completes its task before another thread starts or resumes its work.
3. **Collecting Results**:
   - If a thread performs a task that produces a result, the calling thread can use join() to wait for that result before proceeding with further processing.
4. **Managing Dependencies**:
   - In cases where one thread depends on the output or state of another thread, join() can help manage these dependencies effectively.

Usage:

When you call join() on a thread, the calling thread will be blocked until the thread on which join() was called has completed. The basic syntax is:

```
thread.join();
```

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running.");
        try {
            Thread.sleep(1000); // Simulating some work
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Thread has completed.");
    }
}
public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
        try {
            thread.join(); // Main thread waits for MyThread to finish
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Main thread resumes after MyThread has completed.");
    }
}
```

Key Points:

- If the thread on which join() is called has already completed, the calling thread will continue execution immediately.
- You can also specify a timeout (in milliseconds) in the join() method, which allows the calling thread to wait for a specified period before continuing execution, even if the other thread has not finished.

Overall, the join() method is a useful tool for controlling the flow of execution in multi-threaded applications, ensuring that threads can work together in a coordinated manner.

Q18. What is the difference between wait() and sleep() method?

The wait() and sleep() methods in Java are used for different purposes in thread management and synchronization. Here's a comparison highlighting their key differences:

Aspect	wait()	sleep()
Purpose	Used for inter-thread communication and synchronization. It causes the current thread to wait until another thread invokes notify() or notifyAll() on the same object.	Used to pause the execution of the current thread for a specified period. It does not release any locks held by the thread.
Lock Release	Releases the monitor lock (if the thread holds one) when it goes into the waiting state, allowing other threads to acquire the lock.	Does not release any locks. The thread remains in a sleeping state but holds onto any locks it has.
Usage Context	Must be called from within a synchronized block or method. It operates on an object's monitor.	Can be called from anywhere and does not require synchronization.
Resumption	The thread is resumed only when another thread calls notify() or notifyAll() on the same object it is waiting on.	The thread is resumed automatically after the specified sleep duration has passed.
Parameter	Takes no parameters or can take a timeout value (in milliseconds).	Takes a timeout value (in milliseconds) or two parameters (milliseconds and nanoseconds).

Throwing Exception	Throws IllegalMonitorStateException if not called from a synchronized context.	Throws InterruptedException if the sleeping thread is interrupted.
Example Use Case	Used in scenarios like producer-consumer, where a thread needs to wait for a signal from another thread to proceed.	Used to create a delay, such as throttling the execution of a loop or pausing a thread temporarily.

Example:

wait() Example:

```
synchronized (sharedObject) {
   while (!condition) {
      sharedObject.wait(); // Waits for notify() or notifyAll()
   }
}
```

sleep() Example:

```
try {
   Thread.sleep(1000); // Sleeps for 1 second
} catch (InterruptedException e) {
   e.printStackTrace();
}
```

Summary:

- **wait()** is about inter-thread communication and is used when a thread needs to wait for another thread to notify it, while **sleep()** is a simple delay mechanism that pauses execution for a specified time.
- wait() releases locks and requires synchronization, while sleep() does not release locks and can be called freely.

Q19. Is it possible to start a thread twice?

No, it is **not possible** to start a thread twice in Java. Once a thread has been started and it completes its execution, it cannot be restarted. Attempting to start a thread again after it has finished will result in a **IllegalThreadStateException**.

Explanation:

- A thread moves through various states during its lifecycle: **New**, **Runnable**, **Running**, **Blocked/Waiting**, and **Terminated**.
- When you call the start() method, the thread moves from the **New** state to the **Runnable** state, and eventually runs.
- Once a thread has completed its execution and reaches the **Terminated** state, it cannot be restarted.

Example:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running.");
   }
}
public class Main {
   public static void main(String[] args) {
      MyThread thread = new MyThread();
```

```
        thread.start();  // Thread starts and runs normally
        // Trying to start the thread again will cause an exception
        thread.start();  // Throws IllegalThreadStateException
    }
}
```

Key Point:

- A thread can only be started once. After calling start(), it cannot be started again. If you need a similar task to run again, you must create a **new instance** of the thread.

To reuse the same logic, you could instantiate a new **Thread** object:

```
MyThread thread1 = new MyThread();
thread1.start();  // First thread runs
MyThread thread2 = new MyThread();
thread2.start();  // Second thread runs independently
```

Q20. Can we call the run() method instead of start()? What would be the result of calling run() method instead of start() method?

Yes, you **can** call the run() method directly instead of using the start() method, but doing so will not create a new thread. Instead, it will execute the run() method in the **current thread** (the thread that calls it), and no new thread of execution will be started.

Key Difference:

- **Calling start()**:
  - When you call the start() method, a new thread is created, and the run() method is invoked in that new thread.
- **Calling run() directly**:
  - When you call the run() method directly, it behaves like a normal method call. It does not create a new thread; instead, it runs in the same thread from which it was called.

Example:

Case 1: Using start() (Correct Way)

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running in a new thread: " + Thread.currentThread().getName());
    }
}
public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();  // A new thread is created, and run() executes in the new thread
    }
}
```

Output:

arduino
Copy code
```
Running in a new thread: Thread-0
```
Here, the run() method is executed in a new thread (Thread-0).
Case 2: Calling run() Directly
```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running in the current thread: " + Thread.currentThread().getName());
    }
}
public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.run();  // No new thread is created, run() executes in the main thread
    }
}
```
Output:
```
Running in the current thread: main
```
In this case, the run() method is called directly within the **main thread**, and no new thread is created.

Conclusion:

- **start()**: Creates a new thread and calls run() in that new thread.
- **run() directly**: Executes the run() method on the current thread (without creating a new thread).

If you call run() directly, you lose the benefit of multithreading. The code will still execute, but it won’t execute concurrently in a separate thread.

Q21. What is the purpose of the Synchronized method and Synchronized block?

The **purpose of the synchronized method and synchronized block** in Java is to provide a way to control access to shared resources (such as variables, objects, or methods) by multiple threads in a multi-threaded environment. This ensures **thread safety** by allowing only one thread to access the critical section at a time, preventing race conditions and ensuring data consistency.

1. synchronized Method

A synchronized method ensures that only one thread can execute the method at a time on the same object. It applies a lock on the object that calls the method. Once a thread acquires the lock, other threads attempting to access the method are blocked until the lock is released.
Example:
```
class Counter {
    private int count = 0;
    // Synchronized method ensures only one thread can modify 'count' at a time
    public synchronized void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final count: " + counter.getCount()); // Output: 2000
    }
}
```

In this example, the increment() method is synchronized, preventing race conditions when t1 and t2 try to increment the counter simultaneously.

2. synchronized Block

A synchronized block is a more granular way to achieve synchronization. It allows synchronization on a specific block of code, instead of the whole method. This is more flexible as it lets you synchronize only the critical section that needs protection, which can improve performance if not the entire method requires synchronization.
You can synchronize on any object, allowing you to control which objects' locks to acquire.

Example:

```
class Counter {
    private int count = 0;
    private Object lock = new Object(); // Lock object
    public void increment() {
        // Only this block is synchronized, reducing the scope of the lock
        synchronized (lock) {
            count++;
        }
    }
    public int getCount() {
        return count;
    }
}
```

In this example, only the block of code that modifies count is synchronized, allowing better performance by not locking the entire method.

Key Differences:

Aspect	synchronized Method	synchronized Block
Scope	Locks the entire method.	Locks only the block of code within the

		synchronized section.
Flexibility	Less flexible, as the entire method is locked.	More flexible, allowing finer control over which parts of the method are synchronized.
Performance	Can be less efficient if the whole method doesn't need synchronization.	More efficient, as only critical sections of code are locked, improving performance.
Lock Object	The lock is on the current object (this for instance methods, or the class object for static methods).	You can synchronize on any object, giving greater control over what locks to use.

When to Use:

- Use **synchronized methods** when the entire method needs to be synchronized, such as when modifying shared data across multiple threads.
- Use **synchronized blocks** for fine-grained control, especially when only part of the method needs to be synchronized, improving performance by reducing the scope of locking.

In summary, both the synchronized method and the synchronized block are used to ensure that shared resources are accessed in a thread-safe manner, but the synchronized block provides more flexibility and efficiency.

Q22. What is the difference between notify() and notifyAll()?

In Java, both notify() and notifyAll() are methods used for waking up threads that are waiting for a lock on an object. These methods are part of the **inter-thread communication** mechanism and are used in conjunction with the wait() method to manage synchronization in a multithreaded environment.

Key Differences Between notify() and notifyAll():

Aspect	notify()	notifyAll()
Waking Up Threads	Wakes up **one** randomly chosen thread that is waiting on the object's monitor.	Wakes up **all** threads waiting on the object's monitor.
Number of Threads Notified	Only **one** thread is notified, even if multiple threads are waiting on the monitor.	**All** waiting threads are notified, but only one can proceed once it acquires the lock, while the others wait for it to release the lock.
Use Case	Best used when only **one thread** needs to proceed at a time (e.g., in a producer-consumer scenario where only one consumer should consume).	Best used when **multiple threads** need to be notified and compete for the lock (e.g., broadcasting a signal to multiple waiting threads).
Resource Efficiency	More efficient in situations where waking up only one thread is sufficient.	Can cause unnecessary context switching and overhead if many threads are woken up unnecessarily.
Execution Order	The selection of which waiting thread gets notified is **not guaranteed** and is chosen arbitrarily.	All waiting threads are awakened, but only **one** will proceed after acquiring the lock.

Examples:

Example of notify():

```
class Task {
  public synchronized void doTask() {
        System.out.println(Thread.currentThread().getName() + " is waiting...");
        try {
            wait();  // Thread waits until it is notified
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " has been notified.");
    }
    public synchronized void notifyOneThread() {
        notify();  // Notifies one random waiting thread
    }
}
public class Main {
    public static void main(String[] args) {
        Task task = new Task();
        Thread t1 = new Thread(task::doTask, "Thread-1");
        Thread t2 = new Thread(task::doTask, "Thread-2");
        t1.start();
        t2.start();
        try {
            Thread.sleep(1000);  // Delay to ensure both threads are waiting
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        task.notifyOneThread();  // Only one of the two threads will be notified
    }
}
```

In this example, only **one** of the two waiting threads (Thread-1 or Thread-2) will be notified and proceed after the notify() call.

Example of notifyAll():

```
class Task {
    public synchronized void doTask() {
        System.out.println(Thread.currentThread().getName() + " is waiting...");
        try {
            wait();  // Thread waits until it is notified
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " has been notified.");
    }
    public synchronized void notifyAllThreads() {
        notifyAll();  // Notifies all waiting threads
    }
}
public class Main {
    public static void main(String[] args) {
        Task task = new Task();
        Thread t1 = new Thread(task::doTask, "Thread-1");
        Thread t2 = new Thread(task::doTask, "Thread-2");
        Thread t3 = new Thread(task::doTask, "Thread-3");
        t1.start();
        t2.start();
        t3.start();
        try {
            Thread.sleep(1000);  // Delay to ensure all threads are waiting
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        task.notifyAllThreads();  // All three threads will be notified, but will proceed one by one after acquiring the lock
    }
}
```

In this case, **all threads** waiting on the monitor are notified, but they still need to compete for the lock. Only one thread can proceed at a time, while others will wait for the lock to be released.

When to Use notify() vs notifyAll():

- **notify()**: Use when you want to wake up **one** waiting thread, and that is sufficient for the task (e.g., when a single producer or consumer is enough).
- **notifyAll()**: Use when you want to wake up **all** waiting threads, especially when multiple threads are waiting for the same condition and may compete for a lock (e.g., broadcasting a condition change).

In general, notify() is more efficient in scenarios where only one thread needs to proceed, while notifyAll() is useful when multiple threads should be awakened and allowed to compete for resources.

Q23. Explain deadlock? How can deadlock be avoided?

Deadlock is a situation in multithreaded or multiprocessing systems where two or more threads or processes are blocked indefinitely, each waiting for the other to release a resource. None of the threads can proceed because each holds a resource that the other needs, causing a circular wait condition.

Conditions for Deadlock:

Deadlock occurs when all of the following four conditions hold simultaneously:

1. **Mutual Exclusion**: Only one thread can access a resource at a time.
2. **Hold and Wait**: A thread holding one resource is waiting to acquire additional resources held by another thread.
3. **No Preemption**: Resources cannot be forcibly taken from threads; they must be released voluntarily.
4. **Circular Wait**: A set of threads is waiting for each other in a circular chain. Thread A waits for a resource held by Thread B, Thread B waits for a resource held by Thread C, and so on, until a thread waits for a resource held by Thread A.

Example of Deadlock:

Here’s an example where two threads cause a deadlock by holding locks on different resources and waiting for the other to release their lock.

```
class Resource {
    public synchronized void access(Resource otherResource) {
        System.out.println(Thread.currentThread().getName() + " accessed resource.");
        try {
            Thread.sleep(100);  // Simulate some work with the resource
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        otherResource.access(this);  // Try to access the other resource
    }
}
public class DeadlockDemo {
    public static void main(String[] args) {
        Resource resource1 = new Resource();
        Resource resource2 = new Resource();
        // Thread 1: Lock resource1 and then try to lock resource2
        Thread t1 = new Thread(() -> resource1.access(resource2), "Thread-1");
        // Thread 2: Lock resource2 and then try to lock resource1
        Thread t2 = new Thread(() -> resource2.access(resource1), "Thread-2");
        t1.start();
        t2.start();
    }
}
```

In this example, **Thread-1** locks resource1 and waits for resource2, while **Thread-2** locks resource2 and waits for resource1. This leads to a deadlock as both threads are waiting for each other to release the resource.

How to Avoid Deadlock?

There are several strategies to avoid deadlock in multithreading and concurrent systems:

1. Avoid Circular Wait (Lock Ordering)

- Ensure that all threads acquire resources in a fixed, defined order. If all threads follow the same locking order, circular wait conditions can be prevented.
- Example: Always acquire resource1 before resource2 for all threads.

```
public void avoidDeadlock() {
    synchronized(resource1) {
        synchronized(resource2) {
            // Perform task with both resources
        }
    }
}
```

2. Use tryLock() (Avoid Hold and Wait)

- Use non-blocking attempts to acquire locks, like tryLock() in the java.util.concurrent.locks.Lock class, which attempts to acquire the lock and returns immediately if it's unavailable.
- If a thread cannot acquire a lock, it can release any other resources it holds and try again later, thus preventing the hold and wait condition.

```
Lock lock1 = new ReentrantLock();
Lock lock2 = new ReentrantLock();
public void avoidDeadlockWithTryLock() {
    if (lock1.tryLock()) {
        try {
            if (lock2.tryLock()) {
                try {
                    // Perform task with both locks
                } finally {
                    lock2.unlock();
                }
            }
        } finally {
            lock1.unlock();
        }
    }
}
```
3. Use a Timeout (No Preemption)

- Use timeouts when trying to acquire resources. If a thread cannot acquire a resource within a certain period, it can release any resources it already holds and retry.
- The Lock class in java.util.concurrent.locks allows you to specify a timeout when acquiring a lock.

```
if (lock1.tryLock(1, TimeUnit.SECONDS)) {
    try {
        if (lock2.tryLock(1, TimeUnit.SECONDS)) {
            try {
                // Perform task
            } finally {
                lock2.unlock();
            }
        }
    } finally {
        lock1.unlock();
    }
}
```
4. Request All Resources at Once

- A thread should request and acquire all the required resources at once. If it can't acquire all resources, it releases any it holds and retries. This prevents holding a resource while waiting for another one.

5. Deadlock Detection and Recovery

- This approach is more complex and involves periodically checking for deadlocks in the system. If a deadlock is detected, one of the threads involved is terminated or rolled back to free up resources.

Summary:

- **Deadlock** occurs when multiple threads are blocked, each waiting for the other to release a resource, resulting in an indefinite waiting situation.
- To avoid deadlocks:
    1. Use **lock ordering** to avoid circular waits.
    2. Use **non-blocking** tryLock() to avoid hold-and-wait situations.
    3. Implement **timeouts** when acquiring locks.
    4. Request all required resources **at once**.
    5. Use **deadlock detection** techniques in larger systems to identify and recover from deadlocks.

Each method ensures that the necessary conditions for deadlock do not occur, allowing threads to access shared resources without becoming indefinitely blocked.

Q24. What is Thread Scheduler in java?

The **Thread Scheduler** in Java is a part of the Java Virtual Machine (JVM) responsible for determining which thread from the ready (runnable) state will be executed next by the CPU. In a multithreaded environment, multiple threads may be eligible to run, but since the CPU can execute only one thread at a time (per core), the Thread Scheduler decides the order and time slices for executing each thread.

Key Points About Thread Scheduler:

1. **Non-deterministic Behavior**:
   - The thread scheduler operates based on the JVM's implementation and the underlying operating system, so there is no guarantee about the order in which threads will be executed or how long they will run. This makes thread scheduling **non-deterministic**.
2. **Time-Slicing vs Preemptive Scheduling**:
   - **Time-Slicing**: In time-slicing, each thread gets a fixed amount of CPU time. Once the time slice expires, the scheduler selects another thread for execution. This is often implemented in round-robin scheduling.
   - **Preemptive Scheduling**: In preemptive scheduling, the highest-priority thread is always selected for execution. If a thread with a higher priority becomes runnable, it preempts (interrupts) the currently executing lower-priority thread.
3. **Runnable and Running States**:
   - A thread must be in the **runnable** state to be considered by the scheduler. The scheduler chooses from the set of runnable threads.
   - A thread moves to the **running** state when the scheduler assigns CPU time to it.
4. **Thread Priority**:
   - Java threads have priorities (ranging from MIN_PRIORITY to MAX_PRIORITY), and the scheduler can use these priorities to decide which thread to execute next.
   - Threads with higher priority are generally given more preference, but priority alone does not guarantee that a thread will run immediately.

```
Thread thread1 = new Thread(() -> {

});
thread1.setPriority(Thread.MAX_PRIORITY);
Thread thread2 = new Thread(() -> {
   // Task for thread2
});
thread2.setPriority(Thread.MIN_PRIORITY);
```

1. **OS Dependent**:
   - The actual scheduling algorithm used is dependent on the underlying operating system and the JVM implementation. Different operating systems may have different behaviors for thread scheduling.

How Thread Scheduler Works:

- When a thread calls the start() method, it enters the **runnable state**, and it is now eligible to be picked by the thread scheduler.
- The thread scheduler selects one of the runnable threads based on factors like **thread priority**, **time-slicing**, and **operating system scheduling policies**.
- A thread can yield its execution by calling Thread.yield(), which gives the scheduler a chance to switch to another thread.
- If a thread finishes its execution or goes to a blocked state (e.g., waiting for I/O or acquiring a lock), the scheduler moves it out of the runnable pool and selects another thread for execution.

Example:

```
class MyThread extends Thread {
   public void run() {
      for (int i = 0; i < 5; i++) {
          System.out.println(Thread.currentThread().getName() + " - Priority: " + Thread.currentThread().getPriority());
      }
   }
}
public class ThreadSchedulerExample {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();
      MyThread t3 = new MyThread();
      t1.setPriority(Thread.MIN_PRIORITY);  // Priority 1
      t2.setPriority(Thread.NORM_PRIORITY); // Priority 5
      t3.setPriority(Thread.MAX_PRIORITY);  // Priority 10
      t1.start();
      t2.start();
      t3.start();
   }
}
```

In this example, the thread scheduler will likely give **t3** (with the highest priority) more CPU time than **t1** (with the lowest priority), but the exact order of execution is determined by the JVM and operating system.

Summary:

- The **Thread Scheduler** in Java manages the execution of threads, determining which thread gets CPU time.
- The scheduling mechanism is dependent on the JVM implementation and underlying operating system.
- Thread priorities and scheduling policies can influence which thread is selected, but the behavior is not guaranteed or predictable.
- Java uses **preemptive scheduling** and **time-slicing** to manage threads, though these mechanisms may vary based on the platform.

Q25. What is race-condition?

A **race condition** is a situation in a multithreaded or concurrent system where the outcome of the execution depends on the non-deterministic timing or ordering of the threads or processes. It occurs when two or more threads try to access and modify shared data simultaneously, and the final result depends on the order in which the threads execute.
In a race condition, if multiple threads access and modify a shared resource without proper synchronization, it can lead to unpredictable and incorrect results.

Example of a Race Condition:

Let’s consider a simple example where multiple threads are incrementing a shared counter variable.

```
class Counter {
    private int count = 0;
    public void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
public class RaceConditionDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        // Create two threads that increment the counter
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });
        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("Final Count: " + counter.getCount());
    }
}
```

In this example, the expected final count is 2000 (since both threads increment the counter 1000 times), but due to the **race condition**, the output may be less than 2000. The threads may be reading and writing to the shared count variable at the same time, leading to **lost updates**.

How a Race Condition Happens:

A race condition typically occurs in the following steps:

1. **Thread 1 reads** the value of the shared resource (e.g., count).
2. **Thread 2 reads** the same value of the shared resource.
3. **Thread 1 increments** the value and writes it back.
4. **Thread 2 increments** the old value (because it had read the same value before Thread 1 updated it) and writes it back, **overwriting** Thread 1's update.

This means that an update from one thread can be lost, leading to incorrect results.

How to Prevent Race Conditions:

Race conditions can be avoided by ensuring that only one thread accesses or modifies the shared resource at a time. This can be achieved using **synchronization mechanisms**.

1. Synchronized Methods:

One way to prevent race conditions is to make the method that modifies the shared resource synchronized, ensuring that only one thread can execute it at a time.

```
class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

2. Synchronized Blocks:

Instead of synchronizing the entire method, you can use **synchronized blocks** to protect critical sections of the code where shared resources are accessed.

```
class Counter {
    private int count = 0;
    public void increment() {
        synchronized (this) {
            count++;
        }
    }
    public int getCount() {
        return count;
    }
}
```

3. Using Atomic Variables:

For some cases, Java provides atomic classes, such as AtomicInteger, which can handle the increment operation atomically without the need for explicit synchronization.

```
import java.util.concurrent.atomic.AtomicInteger;
class Counter {
    private AtomicInteger count = new AtomicInteger(0);
    public void increment() {
        count.incrementAndGet();
    }
    public int getCount() {
        return count.get();
    }
}
```

Summary:

- A **race condition** occurs when multiple threads access and modify shared data concurrently, leading to unpredictable and incorrect outcomes.
- Race conditions happen when threads perform operations in an interleaved manner without proper synchronization.
- You can prevent race conditions by using synchronization mechanisms like synchronized methods or blocks, or by using atomic classes (e.g., AtomicInteger).

Q26. What are the states in the lifecycle of a Thread?

Thread Lifecycle States in Java

A thread in Java goes through a series of states during its lifecycle, from its creation to termination. The Java thread lifecycle is managed by the JVM, and a thread can be in one of the following states:

1. **New**
2. **Runnable**
3. **Blocked**
4. **Waiting**
5. **Timed Waiting**
6. **Terminated**

These states are defined in the Thread.State enumeration in Java.

1. New:

- A thread is in the **New** state when it is created but not yet started.
- This happens after the thread object is instantiated, but before the start() method is invoked.

```
Thread thread = new Thread();  // New state
```

2. Runnable:

- A thread moves to the **Runnable** state when the start() method is called.
- In this state, the thread is ready to run but may or may not be running depending on the thread scheduler.
- The thread may transition between **Runnable** and **Running** states frequently, but from the JVM's perspective, both are considered part of the **Runnable** state.

```
thread.start();  // Thread is now in Runnable state
```

3. Blocked:

- A thread enters the **Blocked** state when it tries to access a synchronized block or method that is currently being held by another thread.
- The thread will remain in this state until the lock is released, at which point it will move back to the **Runnable** state.

```
synchronized (lock) {
   // Thread will be in Blocked state if it tries to acquire the lock but another thread holds it
}
```

4. Waiting:

- A thread enters the **Waiting** state when it is waiting indefinitely for another thread to perform a specific action.
- This can happen when a thread calls methods like Object.wait(), Thread.join() (without a timeout), or LockSupport.park().
- The thread remains in this state until another thread wakes it up using methods like notify() or notifyAll().

```
thread.join();  // Thread enters the Waiting state until the other thread finishes
```

5. Timed Waiting:

- A thread enters the **Timed Waiting** state when it is waiting for a specific amount of time.
- This happens when a thread calls methods like Thread.sleep(), Object.wait(long timeout), or Thread.join(long millis).
- Once the specified time period elapses, the thread transitions back to the **Runnable** state.

```
Thread.sleep(1000);  // Thread enters Timed Waiting state for 1 second
```

6. Terminated:

- A thread enters the **Terminated** state when it has completed its execution, either by finishing its run() method or due to an uncaught exception.
- Once a thread reaches this state, it cannot be restarted.

```
public void run() {
   System.out.println("Thread is running...");
}  // After run() method finishes, the thread is in the Terminated state
```

Thread Lifecycle Diagram:

Here's a typical lifecycle of a thread:

1. **New** → (start) → **Runnable**
2. **Runnable** ↔ **Running** (managed by the thread scheduler)
3. **Running** → **Blocked/Waiting/Timed Waiting** (when waiting for resources or events)
4. **Blocked/Waiting/Timed Waiting** → **Runnable** (when the condition is met)

5. **Running → Terminated** (after thread finishes execution)

Summary of Thread States:

State	Description
New	Thread is created but not yet started.
Runnable	Thread is ready to run, may be running or waiting for the CPU.
Blocked	Thread is waiting to acquire a lock held by another thread.
Waiting	Thread is waiting indefinitely for another thread to perform a specific action.
Timed Waiting	Thread is waiting for a specific period of time (e.g., sleep(), join(long)).
Terminated	Thread has finished execution and cannot be restarted.

This thread lifecycle provides a structured way to manage thread execution and behavior in Java.